OCSEGen: Open Components and Systems Environment Generator

Oksana Tkachuk
NASA Ames Research Center
oksana.tkachuk@nasa.gov

Abstract

To analyze a large system, one often needs to break it into smaller components. To analyze a component or unit under analysis, one needs to model its context of execution, called *environment*, which represents the components with which the unit interacts. Environment generation is a challenging problem, because the environment needs to be general enough to uncover unit errors, yet precise enough to make the analysis tractable. In this paper, we present a tool for automated environment generation for open components and systems. The tool, called OCSEGen, is implemented on top of the Soot framework. We present the tool's current support and discuss its possible future extensions.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model Checking

General Terms Verification

Keywords User Specifications, Static Analysis, Code Generation

1. Introduction

When analyzing large systems, one often needs to break them into smaller components or units under analysis. Many analysis techniques require the unit under analysis to be executable, i.e., its execution context or *environment* needs to be modeled.

Environment generation is a problem persistent across different types of program analysis: in unit testing, one has to write test *drivers*, components that make calls to the unit, and *stubs*, simplified implementations of actual components called by the unit; in static analysis, one has to supply analysis results for components that are missing or hard to analyze (e.g., native methods); in modular model checking, one has to write both drivers and stubs.

Environment generation is a challenging problem: the environment needs to be general enough to cover interesting unit behaviors and uncover errors, yet restrictive enough to enable tractable analysis, without being overly restrictive, which may cause the analysis to miss important unit behaviors, including errors.

In this paper, we present the Bandera Environment Generator [7, 9], implemented on top of Soot [6]. The approach evolved based on experience gained while applying it to many case studies from various domains. Recently, the tool has been released under an open source license and given a new name: Open Components and Systems Environment Generator (OCSEGen) [4]. OCSEGen can be used as

- *Enabling* technique: when an open component needs to be closed first, e.g., with a test driver, before a back-end analysis can be run.
- *Performance-enhancing* technique: when a system under test is too large for a back-end analysis, e.g., due to the presence of complex libraries that need to be stubbed out first.

The rest of the paper defines the environment generation problem and presents OCSEGen's architecture, usage and possible future extensions.

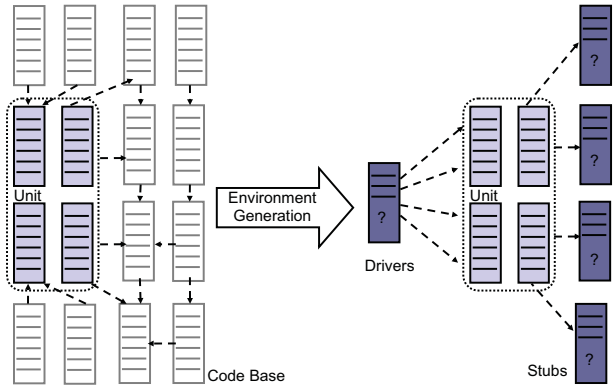

Figure 1. Environment Generation Problem

2. Environment Generation Problem

Figure 1 depicts the environment generation problem. On the left, it shows a codebase as a collection of Java classes. We carve out a unit under analysis as a subset of the entire codebase. On the right, we show that we want to generate the unit's environment consisting of drivers and stubs. We define *drivers* as Java classes that hold a thread of control, i.e., classes containing the `main()` method or classes that extend/implement `java.lang.Thread/Runnable`. The remaining environment classes are called *stubs*. Our experience shows that often, one needs to generate (1) drivers that exercise the unit behavior by performing sequences of actions on the unit and (2) stubs for the library components used by the unit.

Note that Figure 1 shows a common case when drivers make calls to the unit, while stubs are called by the unit. In general,

SOAP'13, June 20, 2013, Seattle, Washington, USA.

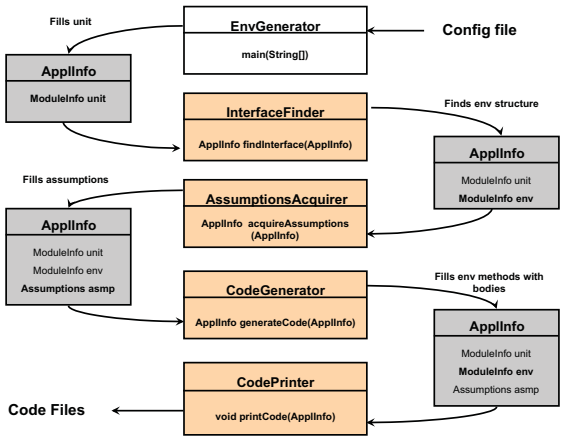

Figure 2. OCSEGen High Level Architecture

the unit-environment interactions may be arbitrary, e.g., stubs may have callbacks to the unit. Interactions between a unit and its environment can be complicated and difficult to analyze: the environment can influence the unit's *control* (e.g., by invoking the unit's methods) and *data* (e.g., by modifying the unit's data flowing into the environment).

When no information is available about the environment behavior, one may start with the most general environment, e.g., a driver that may perform all possible sequences of actions on the unit and stubs that may have all possible side-effects on the unit data. However, such environments are not practical. If some information about the environment behavior is available, it can be used to generate more precise environments. Such information is called *environment assumptions* and it can be mined from a variety of sources, e.g., user specifications, static analysis, run-time analysis, symbolic execution, or learning algorithms.

By default, OCSEGen generates the most general drivers and empty stubs. In addition, OCSEGen has support for the following options:

- *User specifications*: regular expressions and Linear Temporal Logic (LTL) can be used to describe possible sequences of actions that the environment may perform on the unit. This option is useful for driver generation, especially in the absence of the environment driver code.
- *Static analysis*: if the environment code is available, points-to and side-effects analyses can be used to calculate possible side-effects that the environment may have on the unit or environment data. This option is useful for stub generation of library code.

OCSEGen has an extensible architecture that allows more analyses to be plugged in. Next, we describe the tool's architecture and usage.

3. Architecture

Figure 2 shows OCSEGen's architecture. At the high level, OCSEGen consists of four main modules that run in a sequence: `InterfaceFinder`, `AssumptionsAcquirer`, `CodeGenerator`, and `CodePrinter`. The interface finder gathers information about the environment's *structure*: its classes, methods, and fields. The assumptions acquirer gathers information about the environment's *behavior* in the form of environment assumptions. The code generator fills the environment methods' bodies with the behavior according to the environment assumptions. The code printer simply prints out the Java code for the newly constructed environment classes. The tool uses the `ApplInfo` data structure to carry information about the application under analysis from one OCSEGen module to another; each module adds information to the `ApplInfo` instance being passed.

OCSEGen has an extensible plug-in architecture. Each of the modules includes an abstract class that can be extended and customized for specific domains or user needs. `ApplInfo` is also extensible and can encode domain-specific information about the application under analysis. The main class, `EnvGenerator`, reads a configuration file and creates instances of the above types, according to the configuration file. The main class also loads unit classes using Soot, which loads and stores class files as instances of `SootClass`. A `SootClass` has information about the name of the class, its parent, the interfaces it implements, a list of `SootMethods` and `SootFields` of the class. A `SootMethod` has information about the name of the method, its modifiers, its parameter types, return type, and a body, which consists of a list of locals and a list of statements. In this work, we use Jimple, a three-address bytecode representation, which offers typed variables and a limited number of statement kinds. Jimple is a convenient representation for data flow analyses.

Next, we describe each OCSEGen module in more detail.

3.1 Application Information

The `ApplInfo` data structure is used to carry information through the pipeline of the OCSEGen modules. It is used to store the following information: the unit classes, the environment classes, the system class hierarchy, call graph, and environment assumptions. In addition, the class declares abstract methods such as

- `boolean isRelevantType(Type type)`
- `boolean isRelevantClass(SootClass sc)`
- `boolean isRelevantMethod(SootMethod sm)`
- `boolean isRelevantField(SootField sf)`

which are used to specify classes, methods, and fields to *scope* the analysis. For example, `isRelevantClass()` can be used to specify classes in the unit for the driver to work with and `isRelevantField()` can be used to describe specific fields that the side-effects analysis should keep track of.

3.2 Interface Finders

Consider the example in Figure 3, which shows snippets from the customized implementation of the observer-observable pattern[1]. Suppose, classes `Subject` and `Watcher` are considered as a unit under analysis. Then `Buffer`, used to keep track of the registered `Watchers`, becomes part of the environment.

There are two aspects of the unit-environment interface that need to be calculated by OCSEGen: the entry points into the unit under analysis, used to define possible actions that the driver may perform on the unit, and the exit points from the unit into the environment, used to define the structure of the environment stubs. These two aspects are implemented by the following classes: `UnitInterfaceFinder` and `EnvInterfaceFinder`.

`UnitInterfaceFinder` walks over the unit classes and gathers relevant methods and fields. The algorithm uses implementation of `isRelevantClass/Method/Field()` to gather domain-specific

[1] Full example artifacts are available as part of the OCSEGen distribution

```
//unit under analysis
public class Subject extends Observable {
 Buffer obs; //for registered watchers
 public Subject() { obs = new Buffer(); }
 public void add(Watcher o) {
  obs.register(o);}
 public void delete(Watcher o) {
  obs.unregister(o);}
...
}
public class  Watcher implements Observer{
  public boolean registered = false; ...
}
```

```
//environment
public class Buffer extends Vector {
 public void register(Watcher w){
   if(!contains(w)){
     w.registered = true;
     super.addElement(w);
   }
  }
 public void unregister(Watcher w){
  if(super.removeElement(w))
    w.registered = false;
  }  ...
}
```

Figure 3. Customized Observer Implementation (excerpts)

entry points into the unit under analysis. By default all public methods and fields in the unit are treated as possible entry points, e.g., public APIs of `Subject`, `add(Watcher)` and `delete(Watcher)`. This information is later used for driver generation.

One can extend `UnitInterfaceFinder` to collect specific classes, methods, and fields by providing their own implementation of `isRelevant*()` methods. For example, for GUI applications, only special event-handling methods are gathered.

`EnvInterfaceFinder` walks over the unit classes and finds all external references to classes, methods, and fields. For example in Figure 3, all references to `Buffer`, including its methods `register(Watcher)` and `unregister(Watcher)` are considered external. For each external reference, `EnvInterfaceFinder` creates a new `SootClass`, `SootMethod`, or `SootField` and stores them in the `env` field of the `ApplInfo` object. Note that this step discovers the structural information about the environment, not its behavior. Also, the analysis is not scoped at the step, since we need to discover all external references in order to produce stubs that enable compilation of the unit.

Before scanning the unit, the `EnvInterfaceFinder` builds a call graph. This is done to resolve virtual invoke expressions. There are two options available: one based on the Class Hierarchy Analysis (CHA) and one based on a call graph built by Spark [3]. The second one is done using a whole-program analysis and requires presence of the `main` method. Thus, the Spark call graph can be used after the driver generation phase. The CHA-based call graph can be used without a main class, however, it is less precise.

3.3 Assumptions Acquirers

OCSEGen has support for acquiring assumptions from two sources: user specifications and static analysis.

The module for reading user specifications includes a class `SpecReader`, which parses, type checks, and completes a user specification to produce the `Assumptions` object, which encodes the specification information. The left hand side of Figure 4 shows a snippet of the driver assumptions for the observer example. Note that the specification is type checked against the driver actions discovered by the previous module, i.e., `add(Watcher)` and `delete(Watcher)`. If some parameters are omitted, the type checker makes the most general guess to complete the specification. The `Assumptions` object is later used by the driver or stub generators to produce bodies for the environment methods.

The static analysis module includes `SideEffectsAnalyzer`, which walks over the external references discovered in the previous step, performing the interprocedural, compositional, parameterized, flow-sensitive points-to and side-effects analysis [8].

The right hand side of Figure 4 shows an example of the stub assumptions discovered by the side-effects analysis for the `Buffer`'s methods `register(Watcher)` and `unregister(Watcher)`. The analysis calculates that the methods may have side-effects on the methods' parameter of type `Watcher`, due to an assignment to its field, `registered`. When possible, the analysis calculates the right-hand side of the assignment that may cause side-effects. When this is not possible, a special `TOP` value is used to denote an unknown value.

OCSEGen uses Soot's Data Flow Analysis (DFA) framework, which, given an implementation of transfer functions, merge operator, direction of data flow, and initial value, performs a fixed point computation over the control flow graph of each method. OCSEGen uses Soot's intraprocedural DFA framework and extends it by implementing transfer functions for invoke statements. Since OCSEGen calculates modular parameterized information for each environment method, at each call site, it either plugs in the previously calculated information of the called method or, if the method has not been visited yet, it first calculates its summary.

OCSEGen uses domain-specific information, encoded in the `isRelevant*()` methods of the `ApplInfo` instance to (1) scope the call graph based on `isRelevantMethod()`, (2) scope the points-to analysis based on `isRelevantType()`, and (3) scope the side-effects analysis based on `isRelevantField()`. Depending on the implementation of `isRelevant*()` methods, the tool can be tuned to produce side-effects to all unit-type fields or specific fields in the environment. For example, for GUI applications, we defined fields implementing specific features as relevant, e.g., component visibility, enabledness and containment. One can provide their own implementation of `isRelevant*()` methods to collect side-effects to specific objects in the unit or its environment.

3.4 Code Generators

Code generators use information encoded in the `Assumptions` object to build bodies for the environment methods discovered by the interface finders. As mentioned, we use Jimple to perform scanning and static analysis techniques. We extended the Soot framework with classes that represent Java bodies, Java statements and Java expressions. Code generators build Java bodies and attach them to `SootMethod`s of the environment classes.

3.5 Code Printers

The `CodePrinter` is an abstract class that declares methods for pretty printing classes and methods. Currently, OCSEGen provides implementation for `JavaPrinter`, which produces Java code. One can extend the `JavaPrinter` to produce modeling primitives for different analysis frameworks. By default, `JavaPrinter` produces modeling primitives supported in Java PathFinder (JPF) [2], a popular analysis framework for model checking Java programs.

Figure 4 shows snippets of the generated code for a driver (on the left) and stubs (right). We use JPF's special modeling primitives to account for non-deterministic choices in the environment. Such

```
//regular expr assumptions
Main: Subject s; Watcher w1; Watcher w2  #
Register: (add() | delete()) #

//generated code
public class Register extends Thread {
  public Subject s;
  public Watcher w1;
  public Watcher w2;
  ...
 public void run(){
  if(Verify.randomBool()
   s.delete(Verify.randomObject("Watcher"));
  else
   s.add(Verify.randomObject("Watcher"));
   } ...
```

```
//side-effects assumptions
method: <Buffer: void register(Watcher)>
maySummary: {param1.registered=[true]}

method: <Buffer: void unregister(Watcher)>
maySummary:{param1.registered=[false]}

//generated code
public class Buffer extends Vector {
 public void register(Watcher param1){
 // begin may se;
 if(Verify.randomBool()){
  param1.registered=true;
  }// end may se;
} ...
```

Figure 4. Observer Environment Assumptions and Generated Code (excerpts)

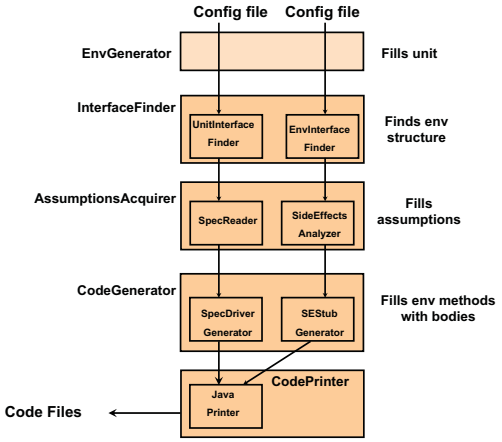

Figure 5. OCSEGen Common Configurations

choices may come from user specifications (e.g., the *or* operator from regular expressions) or the *may* flavor of static analysis. For example, to reflect the possibility of an environment action, it is encoded as `if(Verify.randomBool()){action}`, which forces JPF to explore both branches of the if statement: the one where the action happens and the one where it does not.

4. Usage

OCSEGen has a command line interface. The following command is used to run it: `java EnvGenerator -c <configfile>`, where `<configfile>` is a name of the configuration file, which specifies concrete classes to instantiate for each of the OCSEGen modules, the domain-specific information about the application under test, various options, and unit classes. Based on our experience, the most common configurations include driver generation from user specifications:

```
ApplInfo = DefaultDriverInfo
InterfaceFinder = UnitInterfaceFinder
AssumptionsAcquirer = SpecReader
CodeGenerator = SpecDriverGenerator
specFileName = specs/observer-re.spec
unit = Subject Watcher
```

and stub generation using side-effects analysis:

```
ApplInfo = DefaultStubInfo
InterfaceFinder = EnvInterfaceFinder
AssumptionsAcquirer = SideEffectsAnalyzer
CodeGenerator = SEStubGenerator
unit = Subject Watcher
```

Figure 5 shows the flow of these two approaches. The tool has support for many options, including an option to analyze a unit instead of the environment. This last option is useful when setting up an analysis of libraries without a client.

5. Conclusion and Future Work

In this paper, we presented OCSEGen, a tool for automated environment generation for open components and systems. OCSEGen has been applied to many case studies from various domains. Initially, it was implemented to support driver generation based on all public methods and fields in the unit and to calculate side-effects to unit data [8, 9]. Later, the tool was extended to treat the domain of GUI applications [1] by scoping the unit's entry points to special event-handling methods and scoping the side-effects analysis to keep track of specific fields in the GUI library classes. Then followed the extension to treat web applications [5].

We are currently working on extending the tool for analysis of Android applications. We would also like to add more static analyses, e.g., to calculate control effects for driver generation and to produce reusable library stubs based on modular slicing.

References

[1] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *ASE*, pages 154–163, 2004.

[2] JPF. Website. `http://babelfish.arc.nasa.gov/trac/jpf`.

[3] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[4] OCSEGen. Website. `http://code.google.com/p/envgen/`.

[5] S. P. Rajan, O. Tkachuk, M. R. Prasad, I. Ghosh, N. Goel, and T. Uehara. Weave: Web applications validation environment. In *ICSE Companion*, pages 101–111, 2009.

[6] Soot. Website. `http://www.sable.mcgill.ca/soot/`.

[7] O. Tkachuk. *Domain-Specific Environment Generation for Modular Software Model Checking*. PhD thesis, Kansas State University, 2008.

[8] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *FSE*, Sept. 2003.

[9] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu. Automated environment generation for software model checking. In *ASE*, Oct. 2003.